Inventors' Declaration Submitted Under 37 C.F.R. § 1.131

As a named inventor in U.S. Patent Application No. 10/715,370 (hereafter "the '370 patent application") filed 19 November 2003 and entitled "Method for Providing Physics Simulation Data", I hereby declare that:

1. That the subject matter of the '370 patent application has a date of invention before August 18, 2003.

2. That the date of conception for the subject matter of the '370 patent application was earlier than August 18, 2003 and that, together with my co-inventors, I proceeded forward with reasonable due diligence to the point where the subject matter was constructively reduced to practice by the filing of the '370 patent application, and U.S. Provisional Patent Application No. 60/507,527 filed 2 October 2003 to which the '370 patent application claims domestic priority.

3. That copies of the following documents are attached as supporting evidence of the foregoing statements, and that each of these documents has a date of authorship before August 18, 2003:

a. A twenty-four (24) slide Power Point presentation summarizing the principal architectural aspects of the system and method subsequently disclosed and claimed in the '370 patent application; and,

b. A twenty-two (22) page development documents outlining the principal functional aspects of the system and method subsequently disclosed and claimed in the '370 patent application.

4. That all of the foregoing statements made herein of my own knowledge are true, and that all statements on information and believe are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code, and that such willful false statements may jeopardize the validity of the application or any patent issuing thereon.

Name _____ Date: 2/6/06
Jean Pierre BORDES

Name _____ Date: 2/6/06
Curtis DAVIS

Name _____ Date: 2/6/06
Monier MAHER

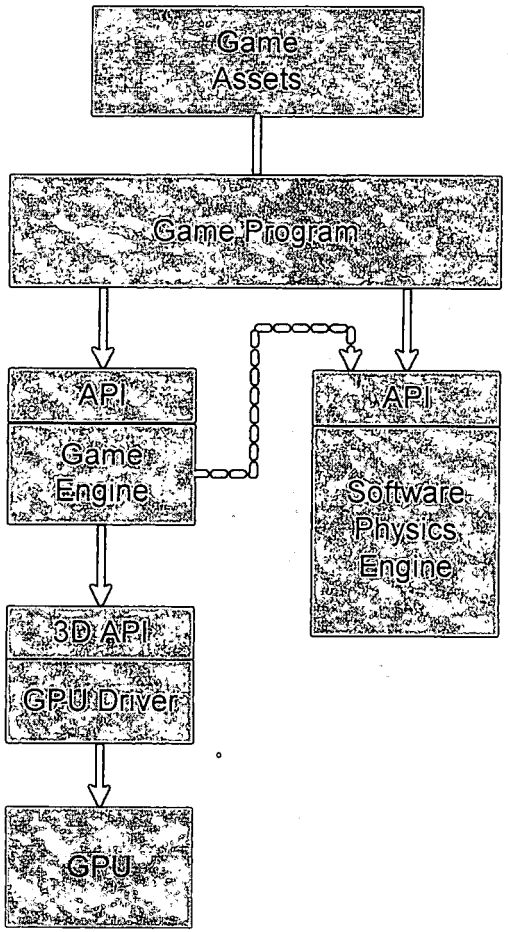
Name _____ Date: 2/6/06
Manju MEGDE

Name _____ Date: 2/6/2006
Otto A. SCHMID

AGEIA

Physics Processing Unit (PPU)

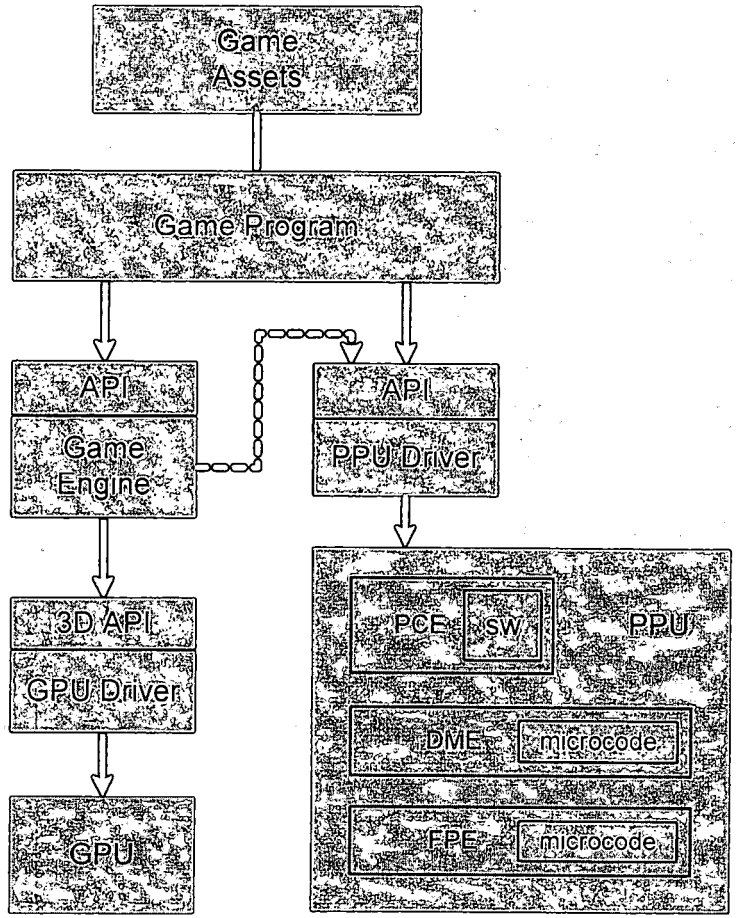
Software Physics



- Parties involved in creating a game:
  - Game Developers
    - Artists (asset creation)
    - Programmers
  - Game Engine Developers
  - Physics Engine Developers
    - MathEngine
    - Havok
    - ODE
  - *Tool Developers*
    - *Discreet (3ds max)*
    - *Maya*

- Today, game physics is completely implemented in software

Ageia PPU



- Havok and MathEngine implement their algorithms on multiple platforms:
  - Windows PC
  - XBox
  - PlayStation 2
  - GameCube
- Ageia partners will port their physics software to the PCE
- Additional differentiation can be obtained by writing custom microcode functions
- The PPU hardware architecture is designed exclusively for running physics simulations

Physics Pain Points

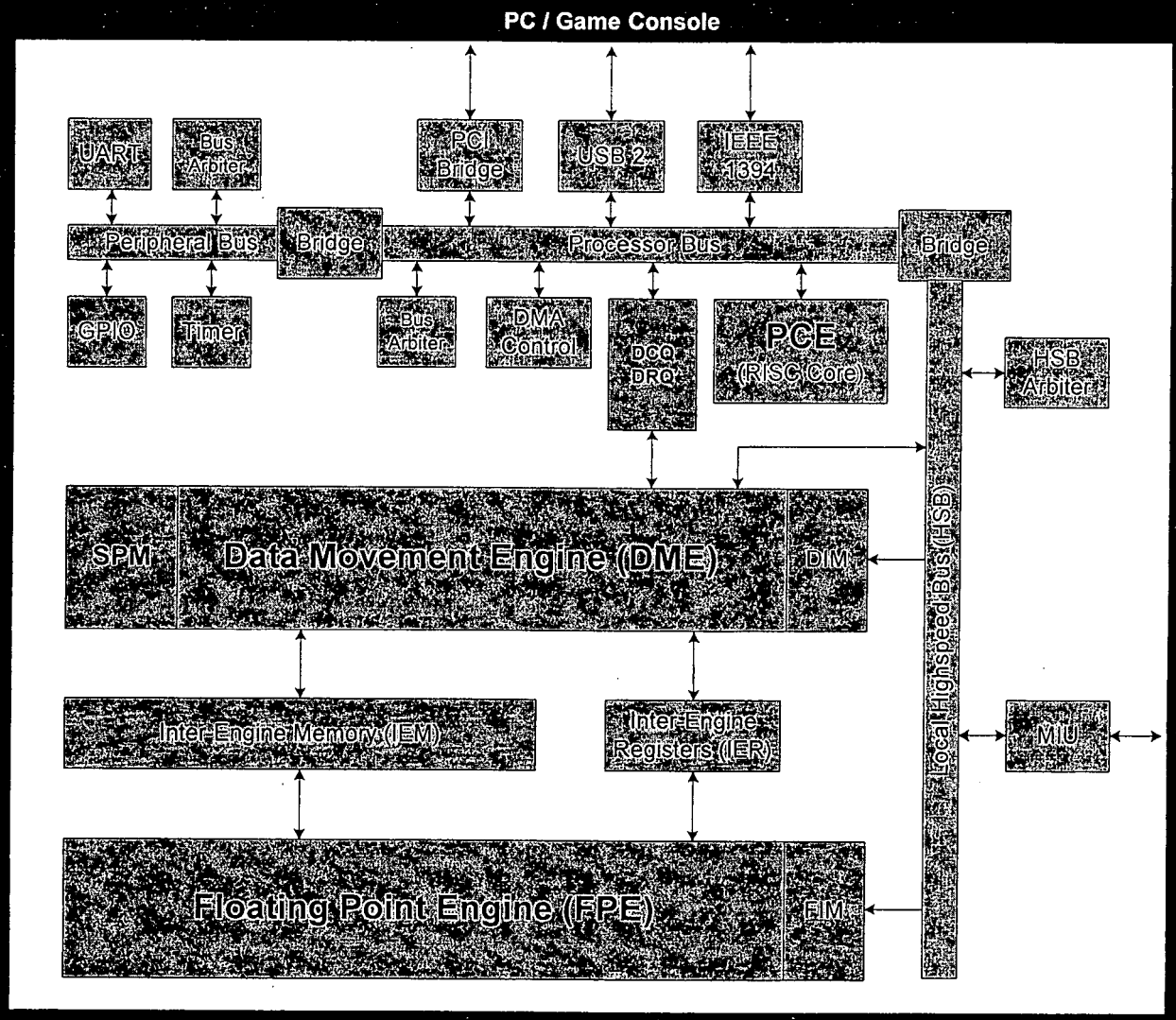
- Software physics engines present severe limitations:
  - Total number of bodies
  - Number of active bodies
  - Number of constraints (i.e.: joints and inter-body contacts)
  - Complexity of collision geometry (# of triangles / body)
  - Complexity of terrain geometry
  - Number of world time steps per second
  - Number of application defined forces per time step
  - Significant CPU power taken away from game and graphics processing

Conventional CPU's

- The primary source of these limitations is the architecture of general purpose CPU's such as the Pentium
  - They are not designed for real-time physics simulation

- CPU Bottlenecks:
  - Limited number of parallel execution units (super-scalar architecture)
  - DRAM latency
  - DRAM bandwidth
  - L1/L2 cache size & set associativity
  - Pipeline flushes
  - General purpose instruction set
  - General purpose architecture

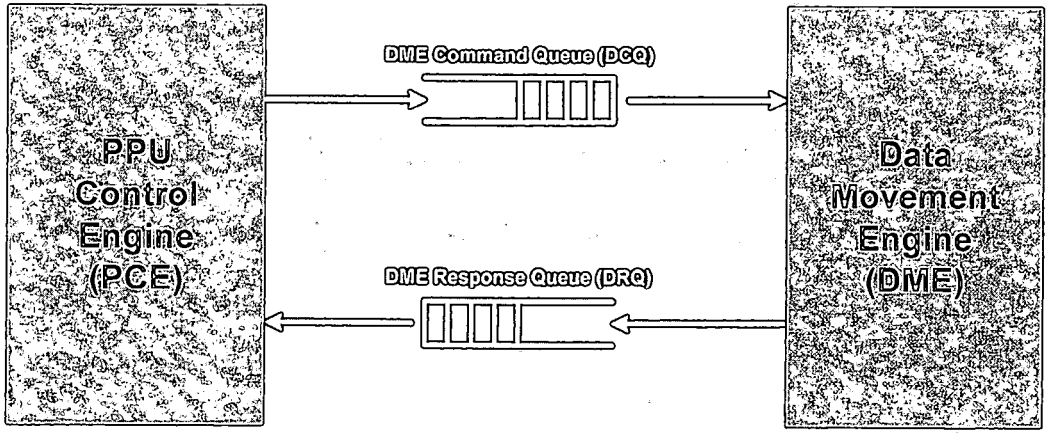
PPU Architecture



Scalable Architecture

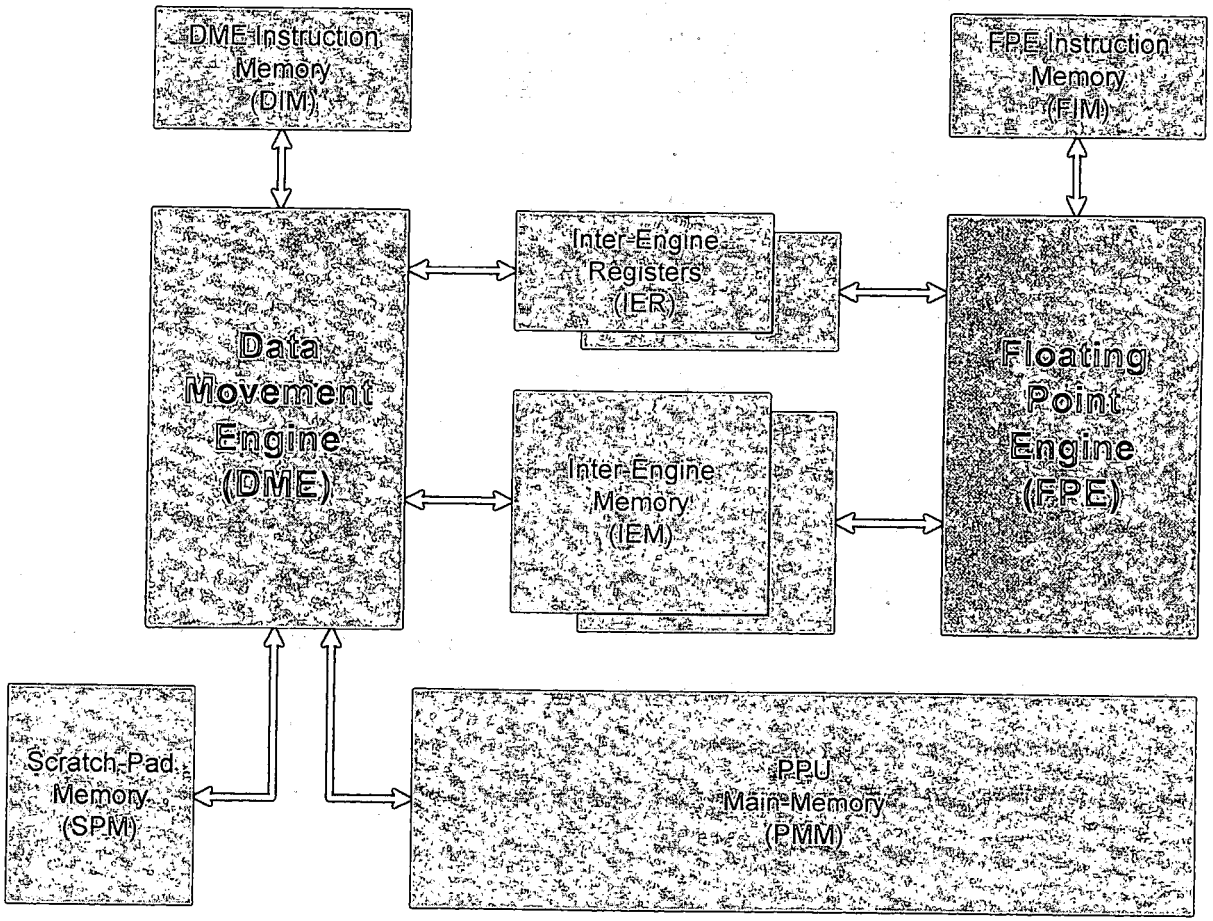
Gene-ration	Sampling	Process	Op. Frequency	Area in mm^2	Performance
1	9/04	130 nm	400/600 MHz	85	154/230 GFLOPs
2	9/05	90 nm	1.080 GHz	160	1.5 TFLOPs
3	9/06	65 nm	1.944 GHz	160	5 TFLOPs

PCE to DME Communication

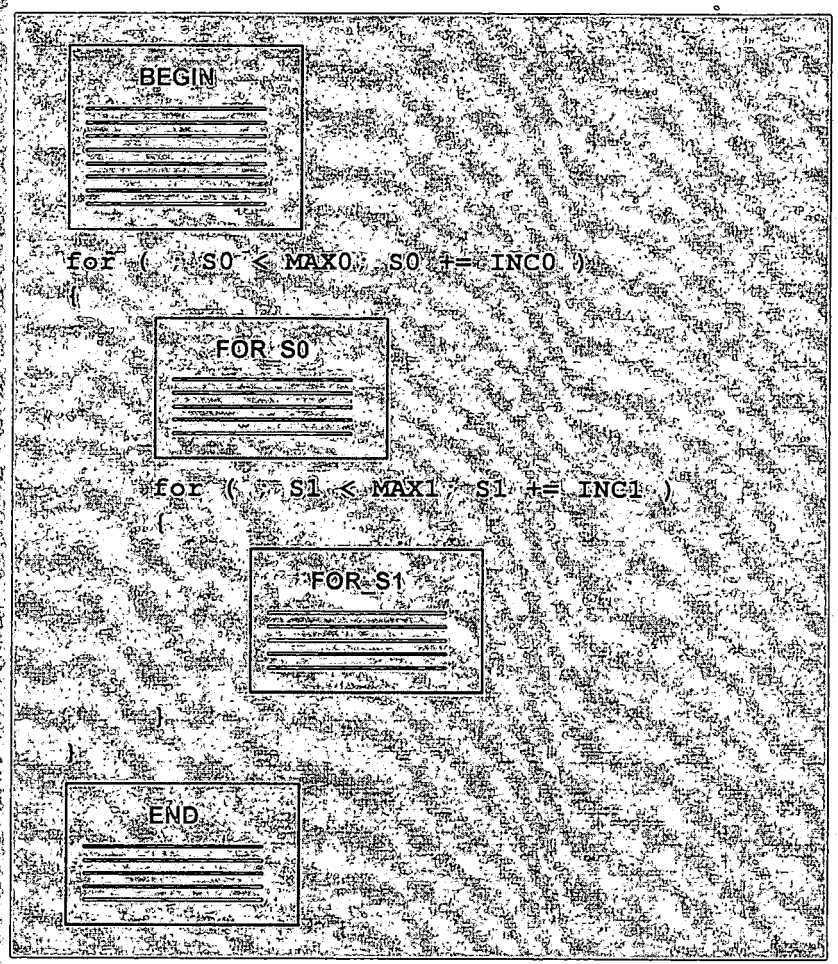


- DCQ and DRQ implemented in dual-ported memory
- DME Control Packet (DCP) contains:
  - DME Program pointer
  - Initial IER register settings
  - DME register settings
  - Program execution flags
- DME Response Packet (DRP) contains:
  - Completion status
  - Program results (from IER)

DME to FPE Communication

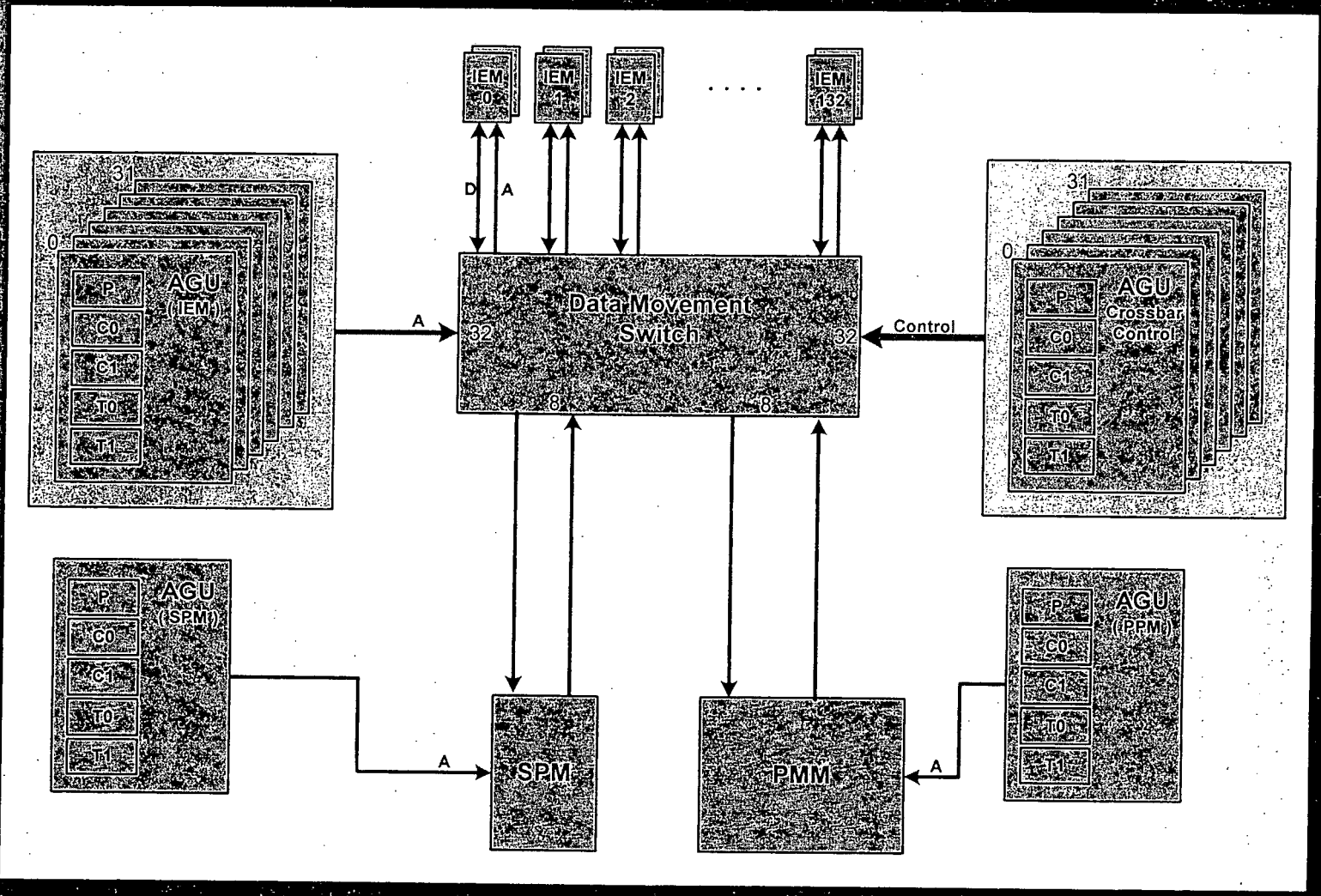


DME Program Flow

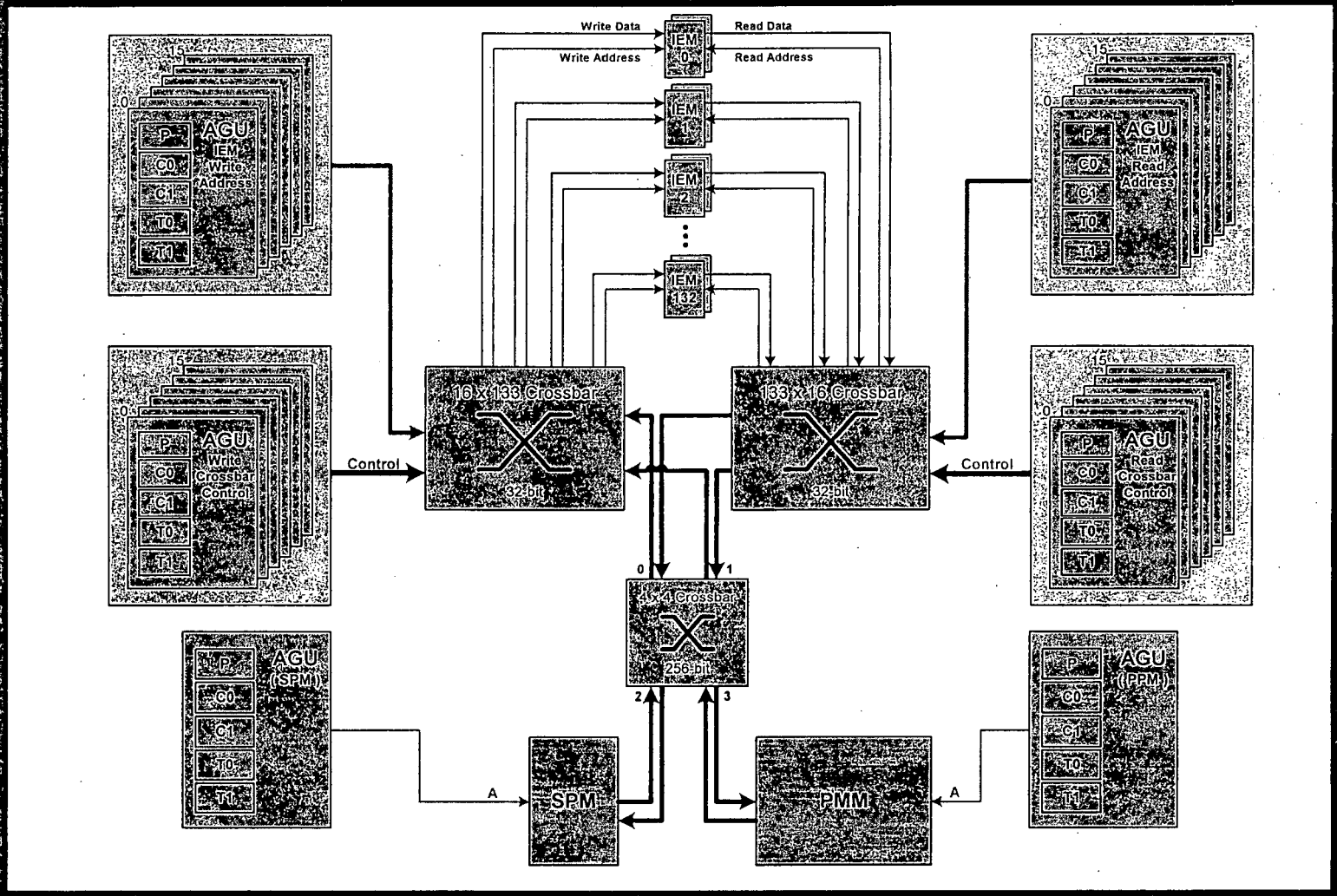


- Data-driven programming model
- S-registers, `MAX[01]`, and `INC[01]` are provided by PCE in DME Control Packet
- 4 sections of "code", each containing:
  - Memory movement parameters
  - SYNC instructions
  - DONE instructions

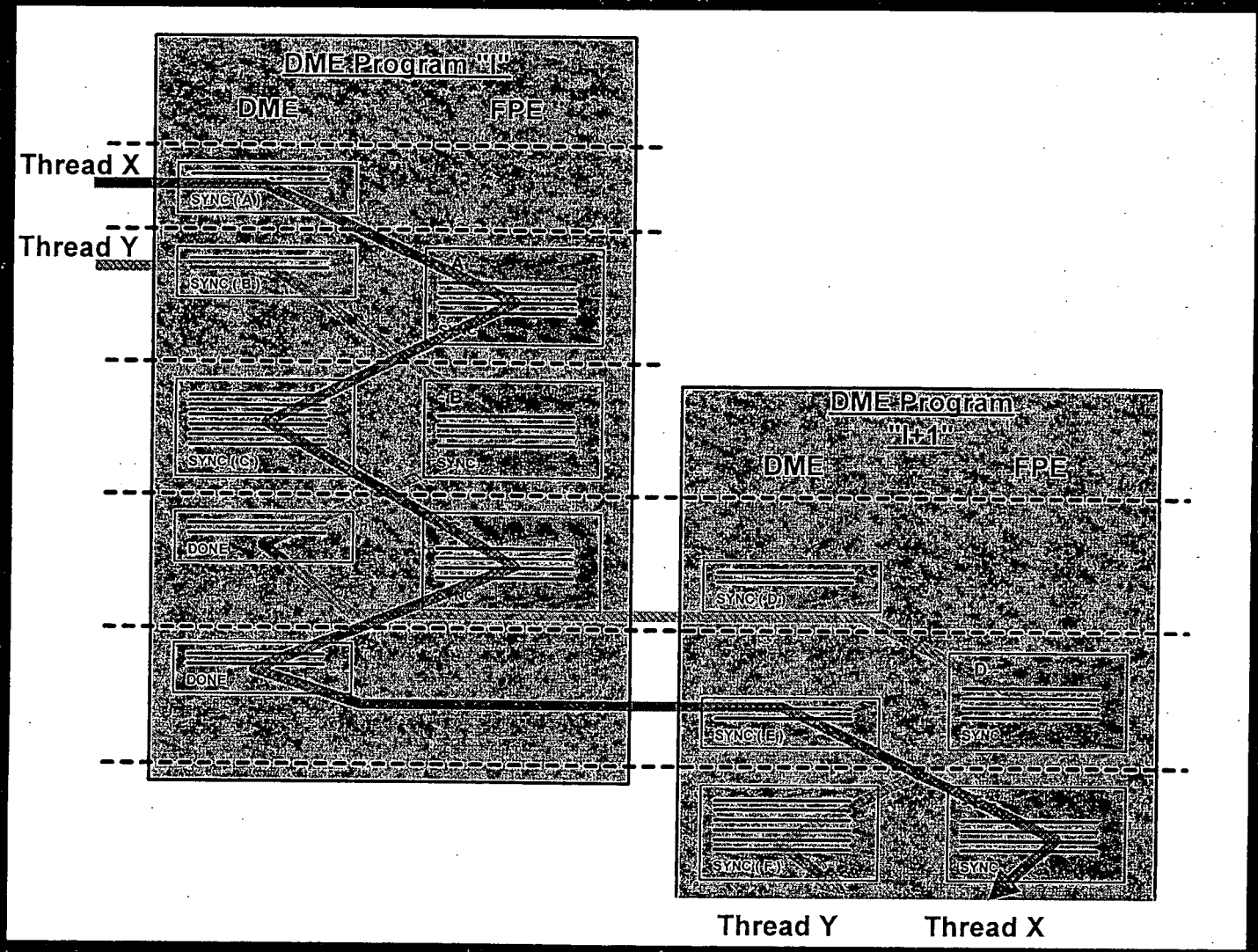
Data Movement in the DME



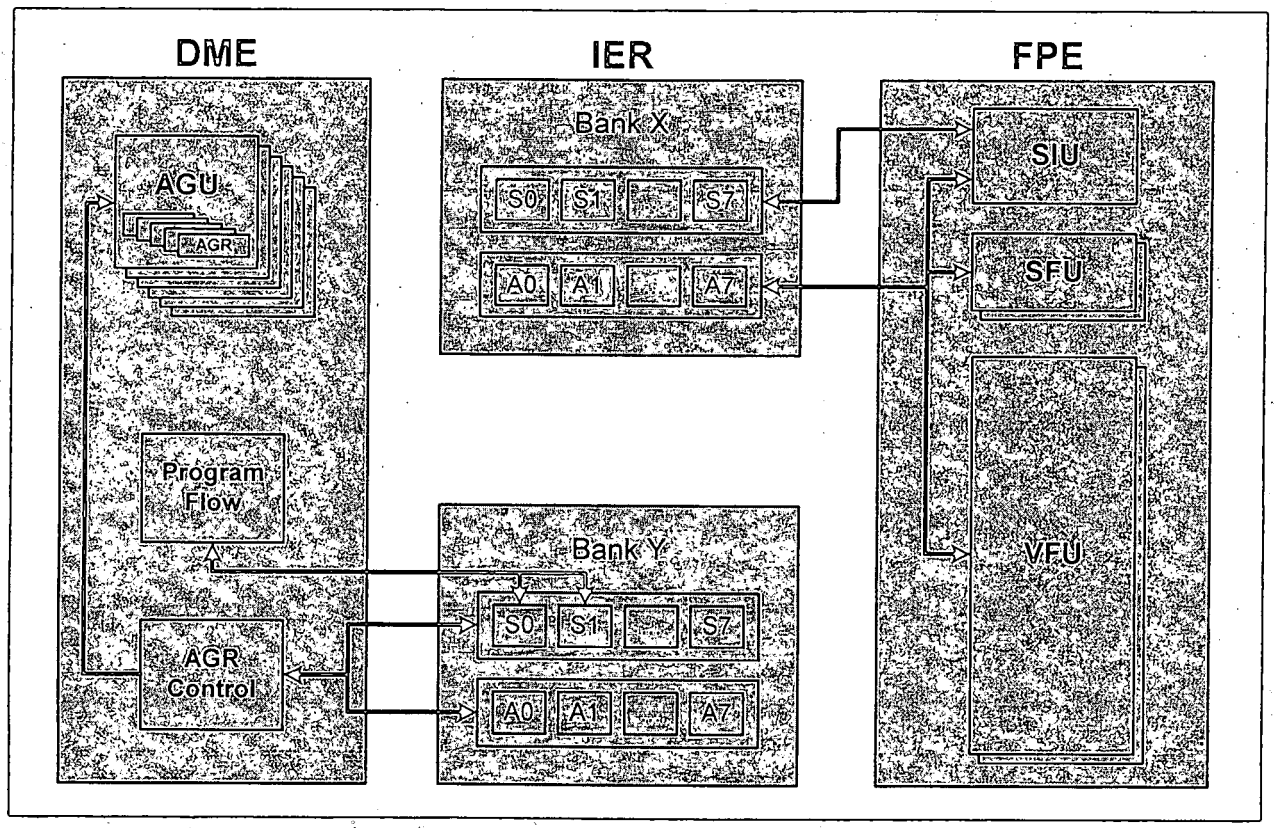
Data Movement in the DME Con'd



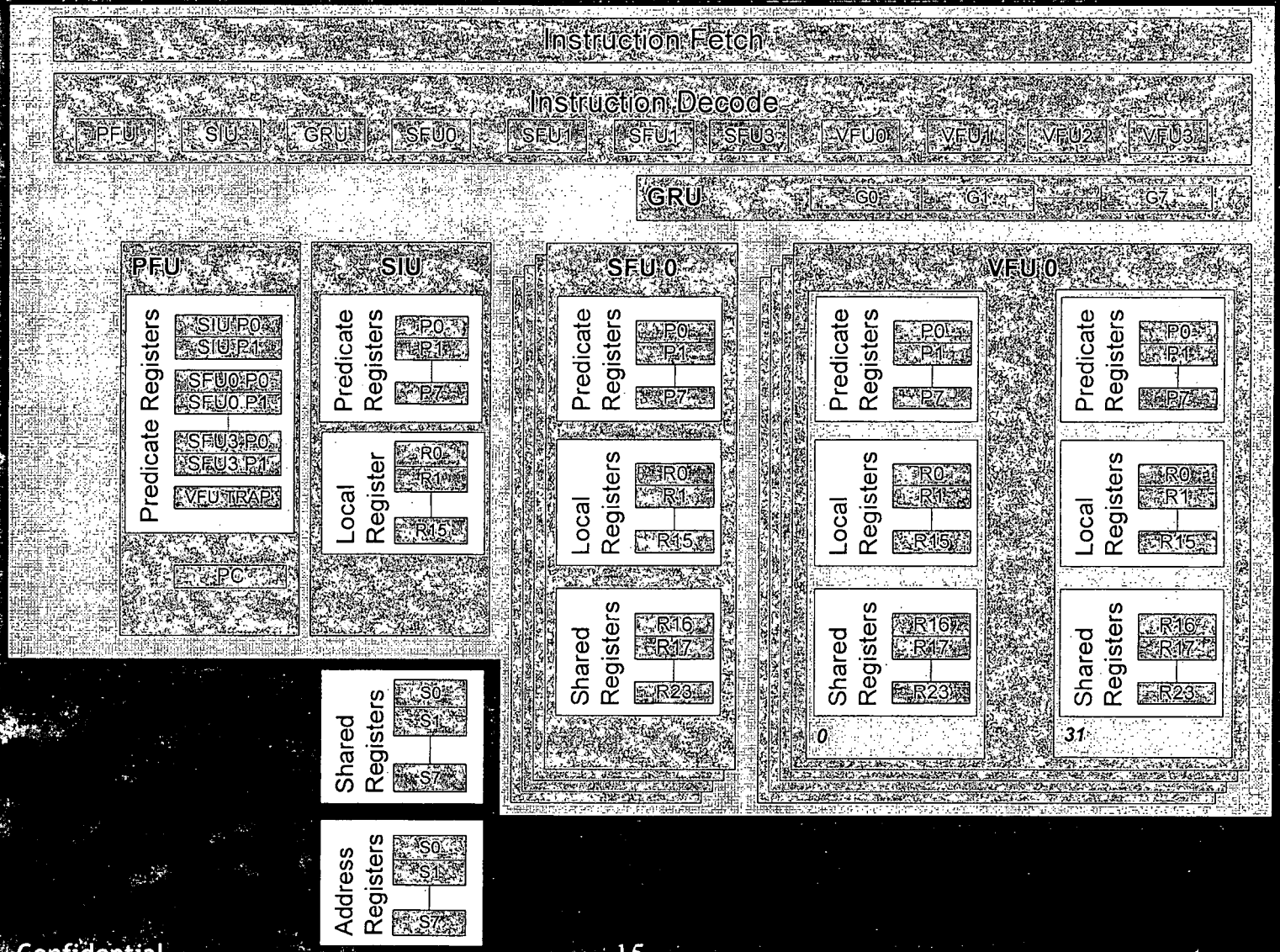
Ultra-Threading



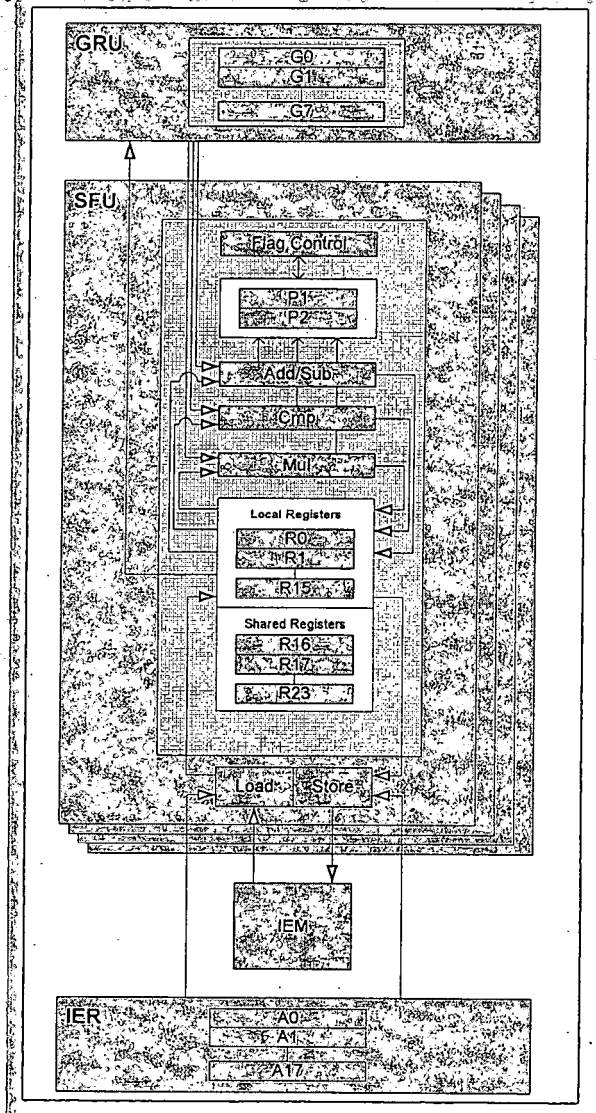
Inter-Engine Registers



Floating Point Engine (FPE)

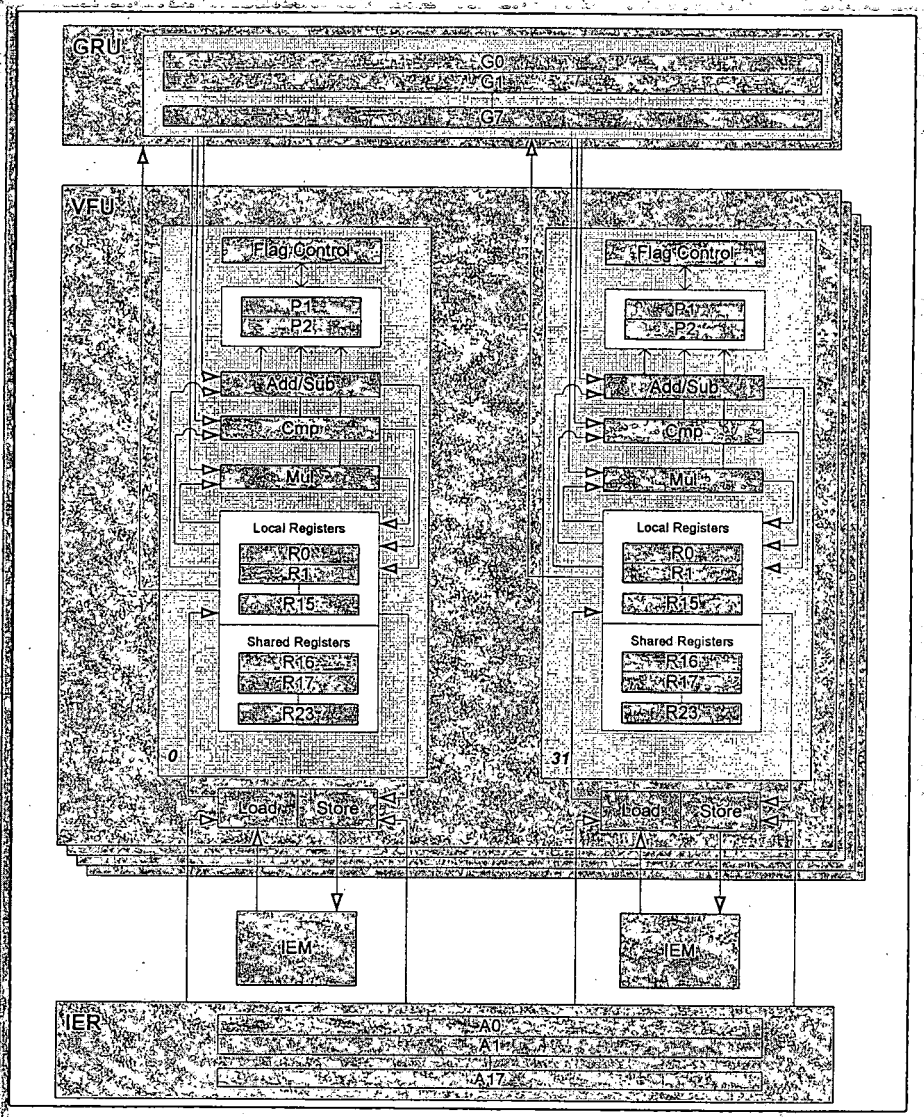


Scalar Floating-point Unit (SFU)



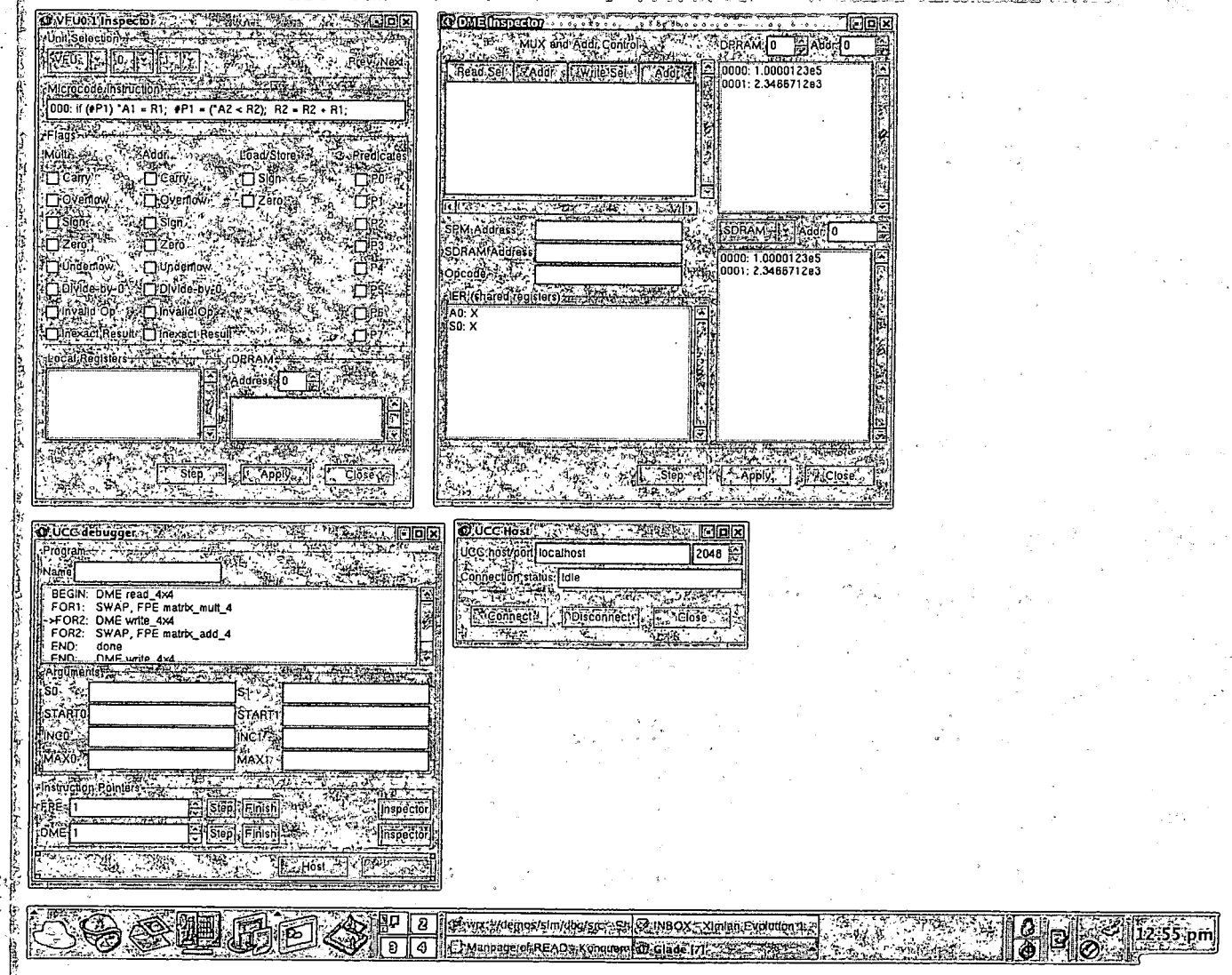
- 32-bit floating-point operations:
  - Add
  - Multiply
  - Divide
  - Square Root
  - Compare
- 16 local registers
- 16 registers shared between SFU's
- Access to GRU and IER's
- 1 IEM Load and 1 IEM Store per cycle
- 3 Arithmetic operations per cycle

Vector Floating-point Unit (VFU)



- 32-bit floating-point operations:
  - Add
  - Multiply
  - Compare
- 16 local registers
- 16 registers shared between VFU's
- Access to GRU and IER's
- 1 IEM Load and 1 IEM Store per cycle
- 3 Arithmetic operations per cycle

SW Development Environment



PPU Innovations

- Optimize design for one application: Physics Simulation
  - Maximize GFLOPS / $$
  - Provide the right mix of GFLOPS and Gbps for physics simulation
  - Avoid wasted GFLOPS by preventing various causes of stalls

PPU Innovations

- PPU architecture designed specifically for running "bottleneck" physics algorithms:
  - Collision detection
    - Axis aligned bounding box intersection tests
    - Multi-level hash table creation and traversal
    - Point-plane tests (witnesses)
    - Triangle mesh intersection tests
  - LCP Solver (Linear Complementarity Problem)
    - Matrix Factorization (LU, LDL^T)
    - Matrix row/column operations (e.g.: pivoting)
    - Matrix transpose
    - Sparse matrices
    - Matrix multiplication
  - Numerical integration of Ordinary Differential Equations
    - Vector dot-product and cross-product

PPU Innovations

- Parallel, task-specific processing modules
  - PCE
    - General purpose processing for miscellaneous operations that are not computationally or bandwidth intensive
    - Off the shelf RISC CPU core
    - Off the shelf programming tools (compiler, debugger, etc.)
  - FPE
    - Massively parallel Floating Point Engine
    - Operates on parallel, ultra-high bandwidth, Inter-Engine Memory
    - No caches!!
  - DME
    - Massively parallel Data Movement Engine (crossbar-based)
    - Data-driven programming model
    - Transfers data between IEM's, and to/from external DRAM
    - Does not block FPE accesses to IEM

PPU Innovations

- Hybrid Vector/VLIW Instruction Set for FPE
  - Vector processing enables hundreds of floating point and data movement operations per clock cycle
  - VLIW architecture allows multiple non-vector operations to occur in parallel with vector operations
  - VLIW instruction word includes instructions for special purpose execution units (e.g.: Global Register Unit, Program Flow Unit)
  - Explicit parallelism in VLIW reduces requirements for hardware pipelining, therefore, more silicon is available for arithmetic units

PPU Innovations

- Large, parallel, on-chip memories (IEM's)
  - Large IEM's eliminate the need for "dumb" caches
    - Explicit control over the contents of on-chip memory
    - Over 2 Terabits/second bandwidth between FPE and IEM
    - Size of L2 cache
    - Latency of L1 cache
    - No "set associativity" limitations
  - Ultra-threading technology
    - Each execution unit has access to two independent IEM's
    - While each execution unit operates on one IEM, the DME operates on the other
    - Zero-latency context switches between IEM banks
    - Ensures that the FPE doesn't stall waiting for data to arrive from memory

PPU Performance

Function	Benchmark	ODE	Ageia	X
Constraint Force Computation	1000 Small Islands	6.7 ms	0.074 ms	91
	200 Rag-Dolls	154 ms	1.4 ms	110
	1 Large Island	144 ms	1.6 ms	90
Collision Detection (5000 rigid bodies)	Zero Collisions	21.5 ms	0.15 ms	143
	2500 Near Collisions	31.0 ms	0.38 ms	82
	2500 Collisions (boxes)	35.6 ms	0.50 ms	71
	2500 Collisions (spheres)	30.1 ms	0.32 ms	94
Numerical Integration	10,000 Bodies	7.57 ms	0.055 ms	138

ageia

PHYSICS PROCESSING UNIT

(PPU)

FUNCTIONAL OVERVIEW

TABLE OF CONTENTS

TABLE OF FIGURES

1 INTRODUCTION

Physics engines used in computer games typically provide the following major pieces of functionality:

- Collision detection
- Collision force computation
- Constraint force computation
- Application force computation
- Rigid body dynamics
- Soft body (particle system) dynamics
- Sensors and event filters

This document will describe how this functionality can be implemented and explain the underlying physical principles. Finally, it will discuss the limitations of a software implementation, and describe how several innovative features of the Ageia Physics Processing Unit (PPU) architecture overcome these limitations.

A physics engine can be conveniently described in terms of its data structures and functional blocks. The data structures are shown in dark gray in Figure 1, whereas the functional blocks are shown in light gray.

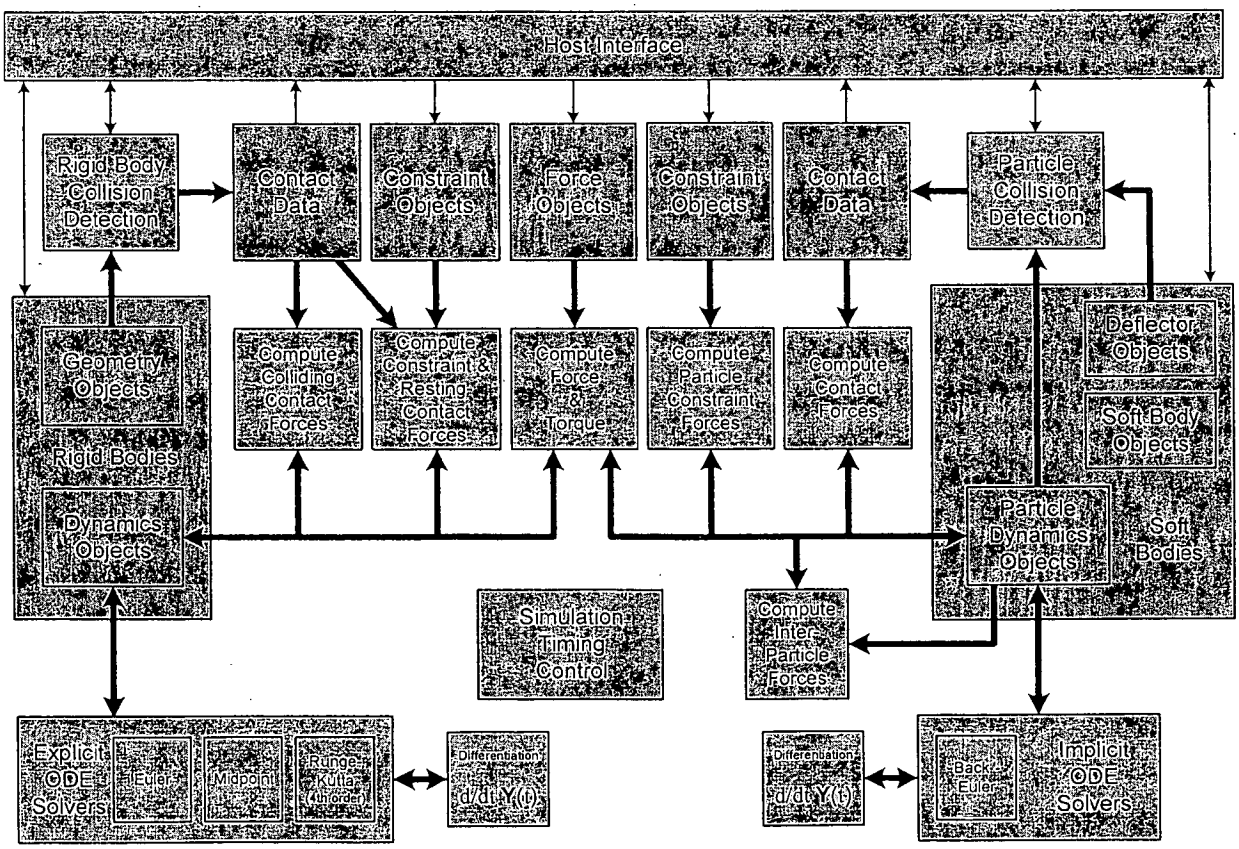


Figure 1: Functional Block Diagram

The rigid and soft body data structures are at the heart of the architecture. They contain all the physical parameters and state information for every simulated object. Physical parameters describe the geometry (which is used for detecting collisions between objects), as well as the kinematics and dynamics (which are used in the physical simulation) of the bodies. They are initially configured by the application, but can also be accessed and modified as the simulation is running. Other data structures that are configured by the application are the force objects and constraint objects. Likewise, these data structures can also be modified as the simulation is running. The contact data structures are automatically re-generated at every simulation time step by the collision detection block, but can be accessed by the application as the simulation is running.

The physics engine consists of four major functional areas: the host interface, collision detection, force computation, and dynamics simulation. Each of these functional areas consists, in turn, of one or more functional blocks.

The host interface provides the application with access to the data structures as well as communication with, and configuration of the engine. It is also responsible for providing event notification to the application (e.g.: monitoring a body for collisions).

Collision detection, just as its name implies, is responsible for detecting collisions between bodies during the course of the simulation. At each time step of the simulation, it updates the contact data structures. The contact force computation unit uses this information to calculate the forces necessary to prevent the bodies from interpenetrating. It can also be accessed by application through the host interface.

Force computation consists of 3 functional blocks which, for each time step, calculate various components of force and torque that are being applied to each rigid body or particle (particles are used in the simulation of soft bodies). Contact forces are computed as the result of contact (collision or resting contact) between bodies. Next, application defined forces are computed by evaluating the force objects configured by the application. Finally, constraint forces are computed in order to guarantee that bodies will not move in ways that would violate the constraints configured by the application through the use of constraint objects. These various forces and torques are added into the force and torque accumulators for each object.

Finally, the dynamics simulation consists of a collection of ODE solvers, a timing control block, and a differentiation block. Several ODE solvers (Explicit Euler, Midpoint, Runge-Kutta) are available to provide different levels of numerical accuracy (at the expense of additional computations). In addition, an implicit integration method (Back Euler) is also required for simulating the particle meshes used in soft bodies. The timing control block is responsible for determining and communicating the size of the next simulation time step. This can be affected by collisions, as well as the error estimate generated by the ODE solver. The differentiation block is responsible for calculating the current time derivative (slope) of each body's state vector. (The state vector, **Y**, contains the current position, rotation, linear momentum, and angular momentum of a rigid body. For particles, it contains only the current position and linear momentum.)

2 PHYSICAL SIMULATION

2.1 PARTICLE SYSTEM DYNAMICS

Particles are objects that have mass, position, and velocity, and respond to forces, but have no spatial extent. Because they are simple, particles are the easiest objects to simulate. Despite their simplicity, particles can be made to exhibit a wide range of behaviors. For example, a wide variety of non-rigid structures can be built by connecting particles with simple damped springs.

The motion of a Newtonian particle is governed by the equation $\mathbf{f} = m\mathbf{a}$, or $d^2\mathbf{x}/dt^2 = \mathbf{f} / m$. This equation involves a second time derivative, making it a second order equation. To handle a second order Ordinary Differential Equation (ODE), it must be converted to a first-order one by introducing extra variables, resulting in a pair of coupled first-order ODE's: $d\mathbf{v}/dt = \mathbf{f} / m$, $d\mathbf{x}/dt = \mathbf{v}$. The position and velocity vectors ($\mathbf{x}$ and $\mathbf{v}$) can be concatenated to form a 6-vector called the state vector ($\mathbf{Y}$). This position/velocity product space is called phase space. A system of n particles can be described by n copies of the equation, concatenated to form a $6n$-long vector. Conceptually, the whole system can be regarded as a point moving through $6n$-space.

A particle simulation involves two main parts – the particles themselves, and the entities that apply forces to the particles. Assuming that appropriate forces can be computed, the simulation consists of using a numerical method for solving initial value problems for ODE's.

2.2 EXPLICIT NUMERICAL INTEGRATION

The simplest method for numerically solving initial value problems for ODE's is the Euler method, which advances a solution from $\mathbf{Y}(t_0)$ to $\mathbf{Y}(t_0 + h)$. The formula for the Euler method is:

$$\mathbf{Y}(t_0 + h) = \mathbf{Y}(t_0) + h\, d/dt\, \mathbf{Y}(t_0)$$

The formula is unsymmetrical: It advances the solution through an interval h, but uses derivative information only at the beginning of that interval. That means that the step's error is only one power of h smaller than the correction.

A better method is the Midpoint method, also known as second-order Runge-Kutta method, which uses the Euler method to take a "trial" step to the midpoint of the interval. Then, it uses the value of both $\mathbf{Y}$ and t at that midpoint to compute the "real" step across the whole interval. First, some notation is necessary:

$$\mathbf{Y}_0 \equiv \mathbf{Y}(t_0)$$
$$f(\mathbf{Y}, t) \equiv d/dt\, \mathbf{Y}(t)$$

Accordingly, the formula for the Midpoint method is:

$$k_1 = h f(\mathbf{Y}_0, t_0)$$
$$k_2 = h f(\mathbf{Y}_0 + \tfrac{1}{2} k_1, t_0 + \tfrac{1}{2} h)$$
$$\mathbf{Y}(t_0 + h) = \mathbf{Y}_0 + k_2$$

Extending this principle further, we get the formula for Runge-Kutta of order 4:

$$k_1 = h f(\mathbf{Y}_0, t_0)$$
$$k_2 = h f(\mathbf{Y}_0 + \tfrac{1}{2} k_1, t_0 + \tfrac{1}{2} h)$$
$$k_3 = h f(\mathbf{Y}_0 + \tfrac{1}{2} k_2, t_0 + \tfrac{1}{2} h)$$
$$k_4 = h f(\mathbf{Y}_0 + k_3, t_0 + h)$$
$$\mathbf{Y}(t_0 + h) = \mathbf{Y}_0 + (k_1 / 6) + (k_2 / 3) + (k_3 / 3) + (k_4 / 6)$$

The fourth-order Runge-Kutta method requires four evaluations of $f(\mathbf{Y}, t)$ per step h, but provides improved accuracy over the Midpoint method and over the Euler method.

2.3 IMPLICIT NUMERICAL INTEGRATION

Instead of assuming (as the Euler method does) that the derivative $d\mathbf{Y}/dt$ throughout the time interval is simply $f(\mathbf{Y}_0)$, let us assume that it is some weighted average of the derivative $f(\mathbf{Y}_0)$ at the beginning of the interval, and $f(\mathbf{Y}(t + h))$ at the end of the interval. Then, we can write the update function:

$$\mathbf{Y}(t + h) = \mathbf{Y}(t) + h [(1 - \lambda) f(\mathbf{Y}(t)) + \lambda f(\mathbf{Y}(t + h))]$$

Where λ is a constant between zero and one. When $\lambda = 0$, this reverts to the ordinary Euler update. Any update of this form where $f(\mathbf{Y}(t + h))$ appears on the right is known as an implicit update formula. When $\lambda = 1$, this equation is known as the backwards Euler or implicit Euler update. Since $\mathbf{Y}(t + h)$ is unknown at the beginning of the step, it may not be clear that the above equation is useful in calculating $\mathbf{Y}(t + h)$. However, around the current state $\mathbf{Y}_0$, we have the approximation:

$$d\mathbf{Y}/dt \approx f(\mathbf{Y}_0) + (\mathbf{Y} - \mathbf{Y}_0) (\nabla f) \mid_{\mathbf{Y} = \mathbf{Y}_0}$$

After substituting, and solving for $\Delta\mathbf{Y}$, we obtain the update formula:

$$\Delta\mathbf{Y} [(1 / \Delta t) I - \lambda (\nabla f) \mid_{\mathbf{Y}=\mathbf{Y}_0}] = f(\mathbf{Y}_0)$$

Computing the update for $\mathbf{Y}$ over a time step now requires solving a linear system. This is the price of using the extra derivative information.

When the above implicit update formula is applied to spring systems, there are some simplifications because the force depends only on the positions of the mass points (particles) and not on their velocities (unless damping forces are used). In this case, the special structure of the problem makes it possible to solve the linear system, represented by a tri-diagonal matrix, with computational work proportional to the number of particles (n). As a result, the method is far faster than the Euler method for simulations of soft body dynamics modeled as systems of point mass particles connected by springs (i.e.: cloth, rope, etc...).

2.4 RIGID BODY DYNAMICS

Simulating the motion of a rigid body is almost the same as simulating the motion of a particle. For a single particle, the state vector $\mathbf{Y}(t)$ is simply defined as:

$$\mathbf{Y}(t) = [\, \mathbf{x}(t),\ \mathbf{v}(t)\,]$$

Where $\mathbf{x}$ represents the position and $\mathbf{v}$ represents the velocity of the particle. If a particle of mass m has a total force $F(t)$ acting on it, then the change of $\mathbf{Y}$ over time is given by:

$$d/dt\ \mathbf{Y}(t) = [\, \mathbf{v}(t),\ F(t) / m\,]$$

As will be discussed later, computing the value $d/dt\ \mathbf{Y}(t)$ is the responsibility of the Differentiation Block.

Rigid bodies, however, are more complicated. In addition to being translated from the origin, rigid bodies can be rotated as well. Rotation is represented as a quaternion, $\mathbf{q}$, in order to prevent numerical drift. Consequently, in addition to linear velocity, rigid bodies can also have angular velocity (ω). The concepts of torque (τ), inertia (I), angular momentum (L) are also required to simulate the motion of a rigid body. The state vector $\mathbf{Y}(t)$ for a rigid body is therefore defined as:

$$\mathbf{Y}(t) = [\, \mathbf{x}(t),\ \mathbf{q}(t),\ P(t),\ L(t)\,]$$

Momentum (both linear and angular) is used in the state vector instead of velocity because angular momentum (a vector which is conservative in nature) is a more convenient representation than angular velocity: If a rigid body is floating through space with no torque acting on it, its angular momentum is constant whereas its angular velocity is not.

The derivative $d/dt\ \mathbf{Y}(t)$ of the state vector for a rigid body is:

$$d/dt\ \mathbf{Y}(t) = [\, \mathbf{v}(t),\ \tfrac{1}{2}\, \omega(t)\, \mathbf{q}(t),\ F(t),\ \tau(t)\,]$$

As is the case in particle dynamics, computing the value $d/dt\ \mathbf{Y}(t)$ for rigid bodies is the responsibility of the Differentiation Block.

3 DATA STRUCTURES

3.1 RIGID BODIES

Rigid body data structures contain all the physical parameters and state information for every simulated object. Physical parameters describe the geometry (which is used for detecting collisions between objects), as well as the kinematics and dynamics (which are used in the physical simulation) of the bodies. They are initially configured by the application, but can also be accessed and even modified as the simulation is running.

3.1.1 Geometry Objects

Geometry objects describe the shape of a rigid body, and are used exclusively for computing collisions with other rigid bodies. They are associated with dynamics objects (below). The following types of geometry objects are supported:

- Simple primitive (sphere, box, plane, cylinder, particle)
- Polygonal mesh (concave, convex)
- Geometry group

A polygonal mesh geometry object contains a pointer to a list of vertices, and a pointer to a list of faces. Faces can be represented as a triangle strip, or as individual triangles. Hierarchies of geometry objects can be created (using the geometry group primitive) to represent complex rigid bodies. All geometry objects include a transform (translation, rotation, scale) that relates the object's local coordinate system to its parent's coordinate system (or to the world coordinate system, if it doesn't have a parent).

The following fields are stored in a geometry object:

- Object type
- Parent geometry object or dynamics object pointer
- Transformation (4x4 matrix)
- Parameters (for simple primitives)
- Triangle vertex list pointer
- Triangle face list pointer

Special "ghost" geometry objects can be created that are not associated with a dynamics object. These geometry objects are only used by the collision detection block, and collisions with these objects do not affect the physical simulation. These objects are useful for generating events that notify the application when a body has moved into or out of a defined space.

3.1.2 Dynamics Objects

Dynamics objects contain all the data associated with a rigid body, other than its shape. This data is initially configured by the application, but is automatically updated at every simulation time step. The following fields are stored:

Physical constants:

- M^{-1} Inverse of Mass
- I_{body}^{-1} Inverse of Inertia Tensor (in body space)

State vector (**Y**):

- x(t) Position
- q(t) Rotation (Quaternions are used to prevent numerical drift)
- P(t) Linear Momentum
- L(t) Angular Momentum

Derived quantities:

- $I^{-1}(t)$ Inverse of Inertia Tensor (in world space)
- v(t) Linear Velocity
- ω(t) Angular Velocity
- R(t) Rotation Matrix

Computed quantities:

- F(t) Force Accumulator
- τ(t) Torque Accumulator

Dynamics objects can be temporarily disabled by the application. While disabled, they do not participate in the physical simulation.

3.2 SOFT BODIES

3.2.1 Soft Body Objects

Soft bodies are used for simulating particle meshes or lattices such as cloth, rope, smoke, water, and fire. Each soft body consists of a mesh or lattice of particles, connected with simple damped springs. Unlike rigid bodies, soft bodies do not require geometry objects, since the geometry of a soft body is implicitly defined by the positions of the particles in the mesh or lattice.

3.2.2 Particle Dynamics Objects

Much like a rigid body, each soft body particle has data associated with it, but since particles are point masses, there is no need for storing moment of inertia, rotation, angular momentum/velocity, or torque. The following fields are stored:

State vector (**Y**):

- x(t) Position
- v(t) Velocity

Other quantities:

- M^{-1} Inverse of Mass
- F(t) Force Accumulator

3.2.3 Deflector Objects

For compatibility with a popular software-based physics engine, collisions are calculated between soft body objects and special deflector objects. Deflectors only represent geometry, and hence do not participate in the physical simulation. The following types of deflector objects are supported:

3.3 FORCE OBJECTS

Force objects are configured by the application in order to apply forces to the rigid and soft bodies that have been created. Although an application can modify force objects at each time-step, even the data-driven force objects are sophisticated enough that for most forces, an object can be created, and allowed operate without intervention for the duration of its existence. Force objects can be used to easily simulate gravity, viscous drag, springs, and spatial Interactions (field forces).

Each force object can be configured to exert a force (possibly producing torque) on a single rigid body (unary force), or equal but opposite forces on two rigid bodies (binary force). A force object can also be configured to exert a force on every rigid body in the simulation.

Force objects can also act on soft bodies. In this case, a force can be made to act on a single particle, every particle in a single soft body, or every particle in every soft body.

3.3.1 Data-Driven Force Objects

Data driven force objects are a simple way for the application to control standard types of forces acting on various bodies. The simplest data-driven force object is the constant force. At each time step, this object will exert a constant force and/or torque on a specified object. A constant force object may be updated periodically (possibly at every time step) by the application, or may be left alone until deleted. Data-driven force objects can also exert forces that are simple mathematical functions of the parameters in the dynamics object (e.g.: position, velocity, angular momentum, etc...).

3.3.2 Procedural Force Objects

For more sophisticated forces, instead of just providing a mathematical function, the application can provide a procedure to compute a force that will be applied to a body (or between bodies). This allows reduced communication with the application at each time step, since the procedural object can calculate the proper force, instead of requiring the application to provide it.

3.4 CONSTRAINT OBJECTS

3.4.1 Rigid Body Constraints

Rigid body constraints allow the application to configure various restrictions on the way rigid bodies move. These constraints are also known as "Joints". The following types of constraints are supported:

- Ball and Socket
- Hinge / Axle
- Slider / Piston
- Universal
- Springs
- Fixed
- Angular Motor

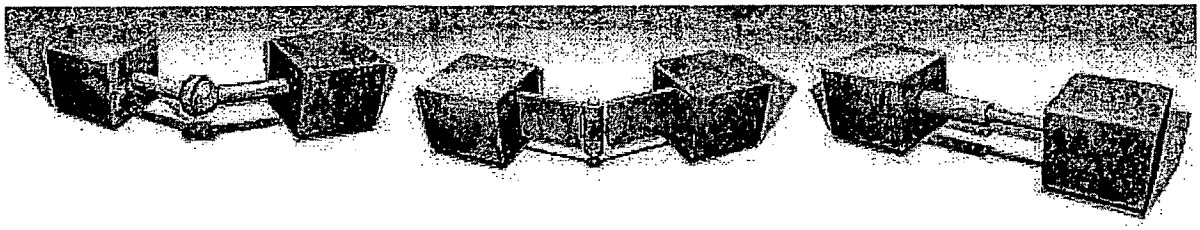


Figure 2: Rigid Body Constraints (Ball & Socket, Hinge, Slider)

Constraint objects allow configuration of limits on the relative motions and orientations of the constrained bodies. These limits allow constraints such as hinges to only twist through a limited angle, or for rag doll limbs to ensure that they always maintain realistic poses. Joints with friction lose energy as the joint is manipulated, so that rotations around constraints eventually come to rest.

3.4.2 Soft Body Constraints

Soft body constraints allow the application to configure various restrictions on the way soft bodies move. The position of individual particles or strips of adjacent particles can be constrained relative to a specified reference frame.

3.5 CONTACT DATA

The collision detection blocks generate contact data at every simulation step. Contact data represents the input to the contact force computation blocks, but can also be accessed by the application, through the host interface.

For rigid bodies, the most common contacts are vertex/face contacts and edge/edge contacts. A vertex/face contact occurs when a vertex of one polyhedron is in contact with a face on another polyhedron. An edge/edge contact occurs when a pair of edged contact. It is assumed in this case that the two edges are not collinear. Vertex/vertex and vertex/edge contacts are degenerate, and require special handling. For example, a cube resting on a table, but with its bottom face hanging over the edge would still be described as four contacts; two vertex/face contacts for the vertices on the table, and two edge/edge contacts, one on each edge of the cube that crosses over an edge of the table.

The contact data structure contains the following information:

- Body "A" (containing vertex)
- Body "B" (containing face)
- P Contact point (world space)
- N Outward pointing normal of face
- ea Edge direction for "A"
- eb Edge direction for "B"
- vf Boolean to identify vertex/face or edge/edge contact

4 FUNCTIONAL BLOCKS

4.1 HOST INTERFACE

The host interface block manages all communication with the application. It is responsible for managing event notification and filtering. This allows the application to be notified only of events that it cares about. It provides the mechanism for the application to create, modify, and delete rigid body, soft body, force and constraint objects. It allows the application to periodically (at each frame time) access all position and orientation data for bodies that have moved.

4.2 SIMULATION TIMING CONTROL

The timing control block is responsible for determining and communicating the size of the next simulation time step. This can be affected by collisions, as well as the error estimate generated by the ODE solver. It communicates with the ODE Solver to determine the error estimate, and if the estimate exceeds a configured threshold, it reduces the time step, and restarts the solver. It also communicates with the Collision Detection unit, and when a collision occurs near the middle of a large time step, it approximates the actual collision time, and backs-up the simulation closer to the time when the two bodies first came into contact.

4.3 COLLISION DETECTION

A lot of research has been done in the field of collision detection, and many good algorithms have been developed. Many algorithms can exploit coherence to reduce the amount of work that must be performed at each time step. (Coherence is the use of information from previous time-step to reduce work.) For example, when processing two objects, A and B, if a separating plane can be found for which all of the vertices of A lie on one side, and all of the vertices on B lie on the other side, the equation of the plane can be stored and used in subsequent time steps to easily verify that the objects have not collided with each other. Additional work only need to be performed if separating plane test fails.

Many algorithms use bounding box hierarchies to reduce the complexity of collision detection processing. Typically, the hierarchy is defined by the application, however, at the cost of some additional processing, it could be created automatically by the physics engine. Various types of bounding boxes can be used, such as Axis Aligned Bounding Boxes (AABB's), Object-aligned Bounding Boxes (OBB's), and spherical bounding boxes.

Another algorithm uses a multi-resolution hash table to detect collisions in O(n). The 3 dimensional world is divided into a regular grid. Lower resolution (larger cell size) grid levels are superimposed on the initial grid. When each object is added to the hash table, a grid level is selected such that the object occupies no more than eight cells (voxels) of the grid. For each occupied cell, a corresponding entry is added to the hash table. The hash function is computed using the X, Y, and Z coordinates of the cell, as well as the grid level. Once all objects are added to the hash table, a second pass is made through all objects, and only objects which are found to occupy the same grid cells are candidates for collision.

4.4 FORCE AND TORQUE COMPUTATION

In a traditional software based physics engine, between each integrator step, the application can call functions to apply forces to the rigid body. These forces are added to "force accumulators" in the rigid body dynamics object. When the next integrator step happens, the sum of all the applied forces is used to push the body around. The forces accumulators are set to zero after each integrator step.

By moving the implementation of the physical simulation into hardware, we free the host CPU from a large computational burden. However, we must still provide the application running on the host with a mechanism for controlling the forces exerted on the various bodies in the simulation. This is accomplished through force objects and the force and torque computation block.

4.4.1 Data-driven force objects

The simplest force objects are the data driven force objects. Whenever the application wishes to apply a force to one or more objects, it creates a force object. If the force is constant or can be expressed as a simple mathematical function of parameters in the dynamics object (such as position or velocity), a data-driven force object can be used. The application identifies one or two bodies that the force should be applied to (e.g.: gravitational attraction, magnetic forces, etc.), or specifies that the force should be applied to all bodies (e.g.: earth's gravity, air resistance, etc.).

4.4.2 Procedural force objects

When more sophisticated forces are required, the application can create procedural force objects. The application provides a procedure that can be executed at each time step to compute the force that should be applied. These procedures can make use of local variables to store data, and can also access parameters in the dynamics object.

4.5 COLLIDING CONTACT FORCE COMPUTATION

Colliding contact occurs when two bodies are in contact at some point p, and they have a velocity toward each other. Colliding contact requires an instantaneous change in velocity. Whenever a collision occurs, the state of a body, which describes both position and velocity (actually the momentum is stored in the state vector, but momentum is a constant function of velocity), undergoes a discontinuity in velocity. The methods for numerically solving ODE's require that the state $\mathbf{Y}(t)$ always varies smoothly. Clearly requiring $\mathbf{Y}(t)$ to change discontinuously when a collision occurs violates that assumption.

We get around this problem as follows. If a collision occurs at time t_c, the ODE solver is instructed to stop (or backup to t_c). Using the state at this time, $\mathbf{Y}(t_c)$, the new velocities of the bodies involved in the collision are computed, and $\mathbf{Y}$ is updated. Then, the numerical ODE solver is restarted, with the new state, $\mathbf{Y}(t_c)$, and simulates forward from t_c.

Consider two bodies, **A** and **B**, that collide at time t_0. Let $p_a(t)$ denote the particular point on body **A** that satisfies $p_a(t_0) = p$. Similarly, let $p_b(t)$ denote the point on body **B** that coincides with $p_a(t_0) = p$ at time t_0. Although $p_a(t)$ and $p_b(t)$ are coincident at time t_0, the velocity of the two points may be quite different. The velocity of the point $p_a(t)$ is:

$$d/dt\ p_a(t_0) = \mathbf{v}_a(t_0) + \omega_a(t_0) \times (p_a(t_0) - \mathbf{x}_a(t_0))$$

In the following equation, $n'(t_0)$ is the unit surface normal. Clearly $\mathbf{v}_{rel}$ gives the component of the relative velocity in the direction of the surface normal:

$$\mathbf{v}_{rel} = n'(t_0) \cdot (d/dt\ p_a(t_0) - d/dt\ p_b(t_0))$$

When $\mathbf{v}_{rel} < 0$, the bodies are colliding. If the velocities of the bodies don't immediately undergo a change, inter-penetration will result. Any force that might be applied at P, no matter how strong would require at least a small amount of time to completely halt the relative motion between the bodies. Therefore, we must use a new quantity J, called an impulse. An impulse is a vector quantity, just like a force, but it has units of momentum. Applying an impulse produces an instantaneous change in the velocity of a body.

4.6 CONSTRAINT AND RESTING CONTACT FORCE COMPUTATION

Whenever bodies are resting on one another at some point p (for example, a particle or rigid body in contact with the floor with zero velocity), they are said to be in *resting contact.* In this case, a force must be computed that prevents the body from accelerating downward. Unlike colliding contact, resting contact does not require a discontinuity in velocity.

Consider a configuration with n contact points. At each contact point, bodies are in resting contact, that is, the relative velocity $\mathbf{v}_{rel}$ is zero (to within a numerical tolerance threshold). We can write the distance between the each pair of contact points at future times $t \geq t_0$ as:

$$d_i(t_0) = n'(t) \cdot (p_a(t) - p_b(t))$$

At each contact point, there must be some force $f_i\, n'_i(t_0)$, where f_i is an unknown scalar, and $n'_i(t_0)$ is the normal at the i-th contact point. The goal is to determine what each f_i is. In computing the f_i's, they must all be determined at the same time, since the force at the i-th contact point may influence on or both of the bodies of the j-th contact point. In order to determine the f_i's, we must write each d/dt $d_i(t_0)$ in the form:

$$d^2/dt^2\, d_i(t_0) = a_{i1}f_1 + a_{i2}f_2 + a_{i3}f_3 + \ldots + a_{in}f_n + b_i$$

The above system of equations can be rewritten in matrix form as:

$$\begin{pmatrix} \ddot{d}_1(t_0) \\ \vdots \\ \ddot{d}_n(t_0) \end{pmatrix} = \mathbf{A} \begin{pmatrix} f_1 \\ \vdots \\ f_n \end{pmatrix} + \begin{pmatrix} b_i \\ \vdots \\ b_n \end{pmatrix}$$

All that remains is to solve for the a_{ij} and b_i terms. For a full derivation, please refer to Rigid Body Simulation by David Baraff, specifically "Part II. Non-penetration Constraints" and "Appendix D". Also, please refer to Fast Contact Force Computation for Nonpenetrating Rigid Bodies by David Baraff.

4.7 ODE SOLVERS

The ODE solver blocks perform numerical integration of ordinary differential equations. Explicit methods are used for rigid bodies whereas Implicit methods are used for the particle systems in soft bodies. Several explicit methods are available, with different levels of accuracy, however, increased accuracy requires additional computation. They support adaptive time-step sizes by, at each step, calculating and sending an estimate of the integration error to the simulation timing control block.

4.8 DIFFERENTIATION

The differentiation block is responsible for calculating the current time derivative (slope) of each body's state vector. (The state vector, **Y**, contains the current position, rotation, linear momentum, and angular momentum of a rigid body. For particles, it contains only the current position and linear momentum.)
This unit calculates: d/dt Y(t) where Y(t) is the state at time "t". The inputs to this block are the state vector and the force and torque accumulators stored in the dynamics object.

Rigid Body:

$$d/dt\ \mathbf{Y}(t) = [\ \mathbf{v}(t),\ \ ½\ \omega(t)\ q(t),\ \ F(t),\ \ \tau(t)\ ]$$

Particle:

$$d/dt\ \mathbf{Y}(t) = [\ \mathbf{v}(t),\ F(t) / m\ ]$$

5 SOFTWARE IMPLEMENTATION

Today, game physics is completely implemented in software. Several commercial physics engines are available, including Havok and MathEngine. These engines have been ported to many platforms, including Windows PC's, Microsoft XBox, Sony Playstation 2, and the Nintendo Game Cube. Open-source physics engines are also available, most notably, Open Dynamics Engine (ODE).

Figure 3, below, shows how various components of a computer game interact with each other. Software components are shown in light gray, artist generated data (assets) in dark gray, and hardware in mid gray.

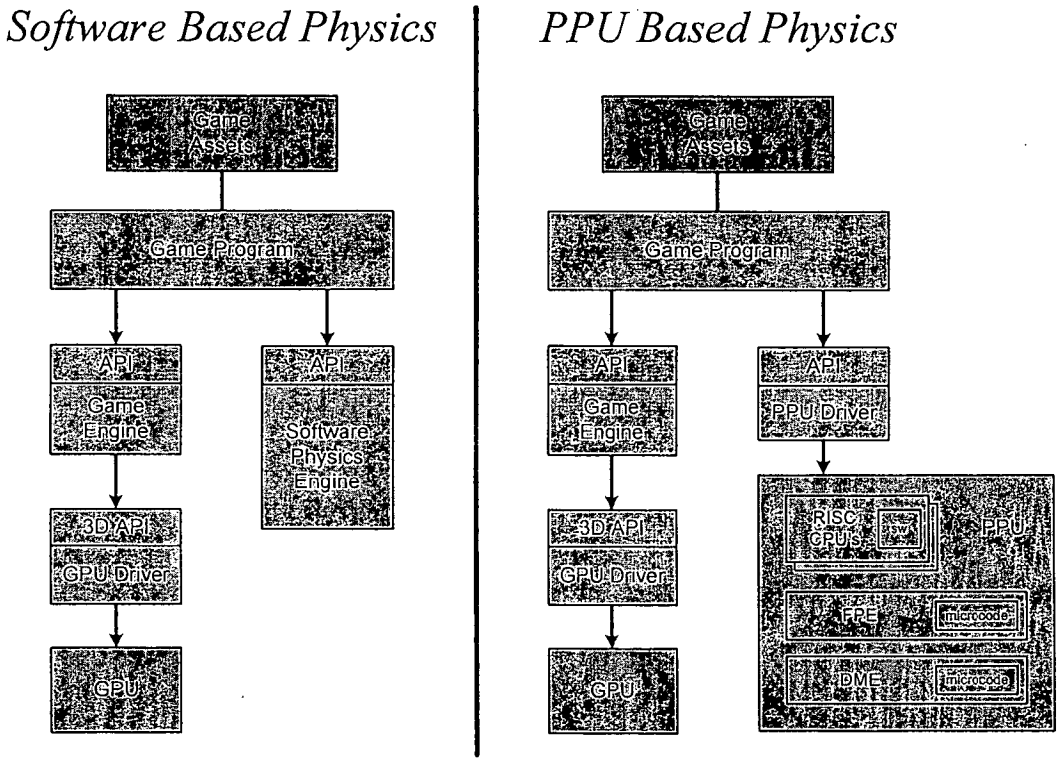


Figure 3: Physics in Computer Games

All physics engines, whether commercial or not, provide similar API's for access to their features. No industry-wide standard API has yet been developed. The physics engine API is frequently accessed directly from the game program, as shown in Figure 3, however, sometimes, the game engine takes over this responsibility, and hides the details of physics processing from the game program.

When physics engine functionality is moved into hardware, a small driver program replaces the software physics engine residing under the API. The driver communicates with the host interface block of the hardware. In the Ageia PPU, this functionality resides on one of the RISC CPU's. Ageia partners, such as MathEngine or Havok will port their engine to the RISC CPU's, and call microcode routines running on the FPE and DME engines. Ageia will provide a library of common linear algebra and physics related algorithms implemented in FPE and DME microcode, however, for added differentiation, we expect that our partners will eventually implement their own performance critical algorithms in FPE and DME microcode.

6 AGEIA PPU INNOVATIONS

6.1 LIMITATIONS OF SOFTWARE PHYSICS

Software physics engines present many severe scale and performance limitations.

- Total number of bodies
- Number of active bodies
- Number of constraints (i.e.: joints and inter-body contacts)
- Complexity of collision geometry (# of triangles / body)
- Complexity of terrain geometry
- Number of world time steps per second
- Number of application defined forces per time step
- Significant CPU power taken away from game and graphics processing

The primary source of these limitations is the architecture of general purpose CPU's such as the Pentium. They are simply not designed for running real-time physics simulations. The same is true for 3-D graphics. In early 3-D computer games, before the advent of accelerated 3-D graphics hardware, rendering was performed entirely in software. However this processing suffered from many of the same bottlenecks as physics engines encounter today.

The following bottlenecks are responsible for the above scale and performance limitations:

- Limited number of parallel execution units (super-scalar architecture)
- Limited DRAM and CPU bus bandwidth
- DRAM latency (cache misses result in stalls)
- L1/L2 cache size & set associativity
- Pipeline flushes
- General purpose instruction set
- General purpose architecture

6.2 PPU INNOVATIONS

The architecture of the Ageia PPU has been carefully chosen to address the specific requirements of physics simulations while avoiding the limitations inherent in conventional CPU's.

6.2.1 Parallel, task-specific processing modules

Extreme parallelism is required to provide the necessary floating point computational capacity required for solving the systems of equations inherent in physics simulation. The Floating Point Engine (FPE) provides this capacity using vector processing units which operate on parallel, ultra-high bandwidth, low latency register files. By avoiding the use of conventional caches and the associated processor stalls, the FPE is able to achieve its theoretical maximum performance, even when operating on large data structures.

In order to keep the register files loaded with the data required by the FPE a massively parallel crossbar-based Data Movement Engine (DME) is provided. It transfers data between register files, as well as to and from DRAM. Because each FPE floating point unit is given two register files, the DME is able to operate in parallel with the FPE without blocking FPE access to the register files.

In addition, two RISC CPU's are provided for general purpose processing for miscellaneous operations that are not computationally or bandwidth intensive. These CPU's use off the shelf cores and come with standard programming tools such as a C compiler, debugger, etc.

6.2.2 Hybrid Vector/VLIW Instruction Sets

The DME and FPE engines both use custom instruction sets which are a hybrid between a vector processing and VLIW architecture. Vector processing is needed to allow hundreds of floating point and data movement operations to be performed per clock cycle. The VLIW instruction word allows multiple non-vector operations to occur in parallel with vector operations. This prevents stalling the vector units while other non-vector operations are executed. Careful analysis of the algorithms required for physics simulation has resulted in an instruction word format that can always provide the necessary non-vector processing in parallel with the vector instructions. For example, the VLIW instruction word includes instructions for special purpose execution units such as the global register unit, and the branching unit.

Explicit parallelism in VLIW also reduces the requirement for hardware pipelining, therefore, more silicon is available for instantiating additional floating point arithmetic units and for larger register files.

6.2.3 Large, parallel, on-chip register files

The use of two banks of large register files eliminate the need for traditional caches. These register files combine the size of a traditional L2 cache with the low latency of an L1 cache. They also provide many times the bandwidth of an on-chip L1 cache, and do not incur any of the limitations of "set associativity".

Rather than using a Least Recently Used (LRU) algorithm and "set associativity" to determine what data should be kept in cache, the DME can be explicitly programmed to load the exact data set that the FPE will need to operate on. Through the use of Ultra-threading technology, the FPE and DME engines exchange register files in a zero-latency context switch. The FPE can immediately begin operating on the newly loaded data, while the DME writes the results of the previous floating point operation(s) to DRAM and loads the data for the next floating point operation(s).

6.2.4 Algorithms implemented specifically for the FPE/DME architecture:

Because the architecture of the Ageia PPU is specifically targeted at running physics simulations, many algorithms can be implemented in FPE and DME microcode such that they take full advantage of the available processing power (115 GFLOPs) and internal bandwidth (4 Tera-bps).

For example:

- LCP Solver (Linear Complementarity Problem)
- Matrix Factorization (LU, LDL^T)
- Matrix row/column operations (e.g.: pivoting)
- Matrix transpose
- Sparse matrices
- Numerical integration of Ordinary Differential Equations